Testing


Cristian Cadar, Daniel Dunbar, Dawson Engler


Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler

Presented by Oren Kishon
9/3/2014
Agenda

• Testing: Introduction
• KLEE + STP: Technical details
• Evaluation
• Related work
• Summary
• Discussion
Agenda

• Testing: Introduction
• KLEE + STP: Technical details
• Evaluation
• Related work
• Summary
• Discussion
Testing

• Purpose :

  • Verifying functional correctness (vs. spec)
  
  • Verifying software completeness - no crashes, memory leaks, assert violations…
Testing

• Purpose:
  • Verifying functional correctness (vs. spec)
  • Verifying software completeness - no crashes, memory leaks, assert violations…
Testing: example

Example [edit]

Consider the program below, which reads in a value and fails if the input is 6.

```
    y = read()
    y = 2 * y
    if (y == 12)
       fail()
    print("OK")
```

- Manual test creation: build test with input 6
- Large number of fail paths?
  - QA person works long hours...
- Test auto-generation
Random input test generation

- ✔ Much more tests generated than manually

- ✘ Error path distribution is not uniform: Boundary values, zero-division...

Back to example: y being a 32 bit int

```python
y = read()
y = 2 * y
if (y == 12)
    fail()
print("OK")
```
Symbolic execution

Consider the program below, which reads in a value and fails if the input is 6.

```plaintext
y = read()
y = 2 * y
if (y == 12)
    fail()
print("OK")
```

- **y is symbolic:** \( y = s \)
- **y = 2 * s**   // still symbolic

Fork execution, add constraints to each path

**true path constraint:** \( 2s = 12 \)

Need constraint solver
Constraint solver

2 * s == 12

CNF: ¬s₁ ∧ ¬s₂ ∧ ¬s₃ ∧ ¬s₄ ∧ s₅ ∧ s₆ ∧ ¬s₇ ∧ ¬0

SAT solver:
satisfiable? -> asserts
instance -> test generate
KLEE: symbolic executor

- Architecture: compiles C code to LLVM byte code. Executes a symbolic interpreter.
- Map LLVM instructions to constraints. Constraint solver: STP.
- Generates executable tests, independent of KLEE.
- Used to check all GNU Coreutils and covered 90% lines: more than 15 year on-going manual test suite - in 89 hours.
Introduction

• Before technical details - any questions?
Agenda

• Testing: Introduction

• KLEE + STP: Technical details

• Evaluation

• Related work

• Summary

• Discussion
Symbolic execution - a deeper look

• Definition: execution state

• Line number

• values of variables (symbolic/concrete): \( x = s_1 \), \( y = s_2 + 3s_4 \)

• Path Condition (PC): conjunction of constraints (boolean formulas) over symbols:
  \( s_1 > 0 \land \alpha_1 + 2s_2 > 0 \land \neg(s_3 > 0) \)
Symbolic execution - a deeper look

- Execute assignment: evaluate RHS symbolically, assign to LHS as part of the state.

- Execute IF (r) / then / else: fork
  - then: PC ← PC ∧ r
  - else: PC ← PC ∧ ¬r

- Termination: solve constraint (supply values for symbols, for test generation)
int y;
int z;
...

int foo(int x) {
    if (x > 0) {
        y = y + x;
    } else {
        y = y - x;
    }
    if (x > 0) {
        z = z - y;
    } else {
        z = z + y;
    }
}

1. int y;
2. int z;
3. ...
4. int foo(int x) {
5.    if (x > 0) {
6.        y = y + x;
7.    } else {
8.        y = y - x;
9.    }
10.   if (x > 0) {
11.      z = z - y;
12.  } else {
13.      z = z + y;
14.  }
15. }

Directed Incremental Symbolic Execution
Suzette Person; Guowei Yang; Neha Rungta; Sarfaz Khurshid
PLDI’11
Execution tree properties

• For each satisfiable leaf exists a concrete input for which the real program will reach same leaf ⇒ can generate test

• PC's associated with any two satisfiable leaves are distinct ⇒ code coverage.
KLEE - usage

Compile C programs to LLVM byte code and run KLEE interpreter with wanted parameters:

$ llvm-gcc --emit-llvm -c tr.c -o tr.bc

$ klee --max-time 2 --sym-args 1 10 10 --sym-files 2 2000 --max-fail 1 tr.bc
KLEE - symbolic execution: tr (Minix)

1: void expand(char *arg, unsigned char *buffer) {
2:     int i, ac;
3:     while (*arg) {
4:         if (*arg == ' \
5:             arg++;
6:         }
7:     }
8:     i = ac = 0;
9:     if (*arg >= '0' && *arg <= '7') {
10:         do {
11:             ac = (ac << 3) + *arg++ - '0';
12:             i++;
13:         } while (i<4 && *arg>='0' && *arg<='7');
14:         *buffer++ = ac;
15:     } else if (*arg != '0')
16:         *buffer++ = *arg++;
17:     else if (*arg == '[') {
18:         arg++;
19:     i = *arg++;
20:     if (*arg++ != '-') {
21:         *buffer++ = '[';
22:         arg -= 2;
23:         continue;
24:     }
25:     ac = *arg++;
26:     while (i <= ac) *buffer++ = i++;
27:     arg++; /* Skip ']' */
28: } else
29:     *buffer++ = *arg++;
30: }

31: int main(int argc, char* argv[]) {
32:     int index = 1;
33:     if (argc > 1 && argv[index][0] == '-') {
34:         ...
35:     }
36:     ...
37:     expand(argv[index++], index);
38:     ...
39: }

3 symbolic arguments

Fork execution

Fork, constraint arg[0]==['

Detect bug (implicit array bounds checking) and generate test: input={"[", "", ""]}

all 37 paths in 2 minutes
KLEE architecture

- Execution state:
  - Instruction pointer
  - Path condition
  - Registers, heap and stack objects
  - Above objects refer to trees of symbolic expressions.
  - Expressions are of C language: arithmetic, shift, dereference, assignment...
  - checks inserted at dangerous operations: division, dereferencing
STP - constraint solver

• A Decision Procedure for Bit-Vectors and Arrays

• “Decision procedures are programs which determine the satisfiability of logical formulas that can express constraints relevant to software and hardware”

• STP uses new efficient SAT solvers.
STP - constraint solver

- Treat everything as bit vectors - no types.
- Expressions on bit vectors: arithmetic (incl. non-linear), bitwise operations, relational operations.
- All formulas are converted to DAGs of single bit operations (node for every bit!)
Refinement formula.

Fig. 1. STP Architecture
Query optimizations

• Constraint solver dominates run time (NP-complete problem in general…)

• Can pre-process calls to solver to make query easier

• Two complicated optimizations (presented next) and other basic ones (later on)
Query optimizations

Constraint independence

- Partition constraint set according to symbols
- Call solver with relevant subset only
- Example: \(\{i < j, j < 20, k > 0\}\). A query of whether \(i = 20\) just requires the first two constraints
Query optimizations

Counter example cache

- Cache results of previous constraint solver results
- If constraint set $C$ has no solution and $C \subseteq C'$, then neither does $C'$
- If constraint set $C$ has solution $s$ and $C' \subseteq C$, then $C'$ has solution $s$
- If constraint set $C$ has solution $s$ and $C \subseteq C'$, then $C'$ likely has solution $s$
State choosing heuristics:

- A big challenge of symbolic executing: path explosion
- Can’t cover all paths: need to choose wisely
- Use different choosing heuristic at each selection (using round robin)
State choosing heuristics:

Random Path Selection

- Maintain binary tree of paths
- When branch reached, traverse randomly from root to select state to execute
- Done to prevent starvation caused by large subtrees (i.e., loops with symbolic condition)
State choosing heuristics:
Coverage-optimize search

- Compute state weight using:
  - Minimum distance to an uncovered instruction
  - Call stack of the state
  - Whether the state recently covered new code
Environment modeling

• Another big challenge of symbolic executing: symbolizing file systems, env. variables, network packets, etc.

• KLEE’s solution: model as much as you can. Modeling means to customize code of system calls (e.g. open, read, write, stat, lseek, ftruncate, ioctl): 2500 lines of modeling code.
Environment modeling

• File system examples

  • Read concrete file with symbolic offset: read() is wrapped with pread()

• Open symbolic file-name:

  • Program was initiated with a symbolic file system with up to N files (user defined).

  • Open all N files + one open() failure
Environment modeling

- How to generate tests after using symbolic env:
  - Except of supplying input args, supply an description of symbolic env for each test path.
  - A special driver creates real OS objects from the description
Other optimizations

• Copy On Write for forking - object level, not page level

• Pointer to many possible objects - branch all

• Query optimizations
  
  • Constraint set simplification: \( \{x<10\}, x==5 \Rightarrow \{x==5\} \)

  • Implied Value Concretization: \( \{x+1==10\} \Rightarrow x = 9 \)
Questions?
Agenda

• Testing: Introduction
• KLEE + STP: Technical details
• Evaluation
• Related work
• Summary
• Discussion
Evaluation - Metrics

- Line coverage, only executable: ELOC percentage
- Doesn’t measure actual conditional paths used
- Used also because the gcov profiler outputs it and its a common tool among testing tools.
Coreutils

- All 89 Coreutils programs ran with command:
  
  ```bash
  ./run <tool-name> --max-time 60
  --sym-args 10 2 2
  --sym-files 2 8
  [--max-fail 1]
  ```
We now give that wants high library coverage can change this setting. A user new statement or branch in the main utility code. A user median coverage per application.

Table 2:

<table>
<thead>
<tr>
<th>Coverage (ELOC %)</th>
<th>Coreutils</th>
<th>OREUTILS</th>
<th>Overall cov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10%</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>10-20%</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>20-30%</td>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>30-40%</td>
<td>4</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>40-50%</td>
<td>5</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>50-60%</td>
<td>6</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>60-70%</td>
<td>7</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>70-80%</td>
<td>8</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>80-90%</td>
<td>9</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>90-100%</td>
<td>10</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

The last rows shows the aggregate coverage across all tools of 84.5%. We get the lines in each tool (median: 94.7%), with an overall coverage results. On average our tests cover 90.9% of the executable lines (ELOC) including library code the tool calls.

While relatively small, the tools are not toys — the small-size of code (ELOC), including library code the tool calls.

Reporting results for the constraint-based execution on a small number of hand-selected benchmarks. Reporting results for the foreign tool where the tool for about 60 minutes (some finished before this limit, others after). For other tools which achieve line coverage in the given ranges for the test five have between 2K and 3K ELOC, over half (52)

Figure 4 breaks down the tools by executable lines size of arguments and files. We found this easy to do, consulted the man page and increased the number and cheating through the use of fragile optimizations.

Almost all tools were tested using the same command arguments explained in §2.1):

- `./run <tool-name> --max-time 60`
KLEE vs. manual suite

<table>
<thead>
<tr>
<th>Coverage (w/o lib)</th>
<th>Coreutils</th>
<th>Busybox</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>klee tests</td>
<td>Devel. tests</td>
</tr>
<tr>
<td>100%</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>90-100%</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>80-90%</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>70-80%</td>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>60-70%</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>50-60%</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>40-50%</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>30-40%</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>20-30%</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>10-20%</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>0-10%</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

**Overall cov.** 84.5% 67.7% 90.5% 44.8%
**Med cov/App** 94.7% 72.5% 97.5% 58.9%
**Ave cov/App** 90.9% 68.4% 93.5% 43.7%

KLEE vs. Manual (ELOC %)

\[
\frac{(L\text{KLEE} - L\text{man})}{L\text{total}}
\]
output tests of bugs

Since 1992

```
paste -d\\  abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
 tac -r t3.txt t3.txt
 mkdir -Z a b
 mkfifo -Z a b
 mknod -Z a b p
 md5sum -c t1.txt
 ptx -F\\  abcdefghijklmnopqrstuvwxyz
 ptx x t4.txt
 seq -f %0 1
```

```
t1.txt: "\t \tMD5(
 t2.txt: "\b\b\b\b\b\b\b\b\b\b\t"
 t3.txt: "\n"
 t4.txt: "a"
```
Observation: random quickly gets the cases it can, and then revisits them over and over
Program equivalence

• Needed in:
  • standard implementation
  • New version testing
Program equivalence

Need to manually wrap programs:

```c
1: unsigned mod_opt(unsigned x, unsigned y) {
2:   if((y & -y) == y) // power of two?
3:       return x & (y-1);
4:   else
5:       return x % y;
6: }
7: unsigned mod(unsigned x, unsigned y) {
8:   return x % y;
9: }
10: int main() {
11:   unsigned x,y;
12:   make_symbolic(&x, sizeof(x));
13:   make_symbolic(&y, sizeof(y));
14:   assert(mod(x,y) == mod_opt(x,y));
15:   return 0;
16: }
```
Program equivalence

Coreutils vs. Busybox

Interesting mismatches:

<table>
<thead>
<tr>
<th>Input</th>
<th>BUSYBOX</th>
<th>COREUTILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm t1.txt t2.txt</td>
<td>[does not show difference]</td>
<td>[shows difference]</td>
</tr>
<tr>
<td>tee -</td>
<td>[does not copy twice to stdout]</td>
<td>[does]</td>
</tr>
<tr>
<td>tee &quot;&quot; &lt;t1.txt</td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
<tr>
<td>cksum /</td>
<td>&quot;4294967295 0 /&quot;</td>
<td>&quot;/: Is a directory&quot;</td>
</tr>
<tr>
<td>split /</td>
<td>&quot;/: Is a directory&quot;</td>
<td>&quot;missing operand&quot;</td>
</tr>
<tr>
<td>tr [ 0 ′′&lt;′′ 1 ]</td>
<td>[duplicates input on stdout]</td>
<td>&quot;binary operator expected&quot;</td>
</tr>
<tr>
<td>sum -s &lt;t1.txt</td>
<td>&quot;97 1 -&quot;</td>
<td>&quot;97 1&quot;</td>
</tr>
<tr>
<td>tail -2l</td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td>unexpand -f</td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td>split -</td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td>ls --color-blah</td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
</tbody>
</table>

Input: t1.txt: a    t2.txt: b
Agenda

- Symbolic testing: Introduction
- KLEE + STP
- Metrics, experimental methods, results
- Related work
- Discussion
Related work

• Similar to KLEE path choose heuristic: generational search (Godefroid, P., Levin, M. Y., And Molnar, D. Automated whitebox fuzz testing)

• Give score to states according to line coverage they done.

• But uses random values when symbolic execution is hard (environment interfacing)
Related work

• Concolic (concrete/symbolic) testing: Run on concrete random inputs. In parallel, execute symbolically and solve constraints. Generate inputs to other paths than the concrete one along the way.


  • Sen, Koushik; Darko Marinov, Gul Agha (2005). "CUTE: a concolic unit testing engine for C"
Agenda

• Symbolic testing: Introduction
• KLEE + STP
• Metrics, experimental methods, results
• Related work
• Discussion
Discussion

• Code coverage is not good enough as a metric. Path coverage is preferred (admitted in the paper)

• Symbolic environment interaction - how reliable can the costume modeling really be? think about concurrent programs, inter-process programs, etc.

• What is more commonly needed - functional testing or security/completeness/crash testing?
KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment

Raimondas Sasnauskas*, Olaf Landsiedel*, Muhammad Hamad Alizai*, Carsten Weise‡, Stefan Kowalewski‡, Klaus Wehrle*

*Distributed Systems Group, ‡Embedded Software Laboratory RWTH Aachen University, Germany

• Sensor networks: network of nodes with unreliable, resource-constrained devices

• On comm loss: hard to find/fix

• Packet loss/corruption, often reboots
KleeNet

- Node model - same as Klee’s environment model. Focuses on TCP failures (invalid packets, etc)

- **Network model**: Holds status of network and packet passing. Injects network wide failures.

- Essentially its a testing tool for distributed systems
Symbolic protocol execution

Injected node reboot - creates new node!
KleeNet

- Insight - after all, complicated systems need customizing tests...